Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE - SEMESTER–III (NEW) EXAMINATION – SUMMER 2024

Subject Code:3130702 Date:19-07-2024

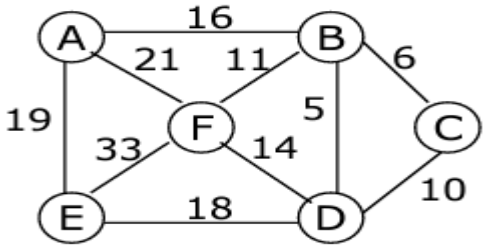
Subject Name: Data Structures

Time:10:30 AM TO 01:00 PM Total Marks:70

Instructions:

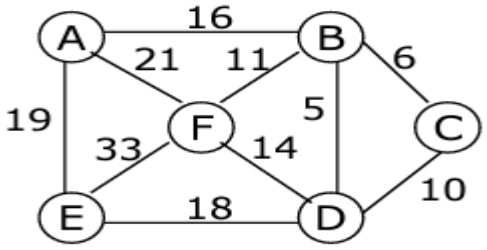
1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			MARKS
Q.1	**(a)**	Explain data structure. Enlist and explain the importance of data structure.	**03**
	(b)	Define time and space complexity. Derive time complexity of merge sort.	**04**
	(c)	1) Evaluate the postfix expression in tabular forms: 2 5 3 - * 8 / 4 +	**03**
		2) Evaluate the prefix expression in tabular forms: / 7 * 1 + 4 – 6 3	**04**
Q.2	**(a)**	Explain Tower of Hanoi with suitable example.	**03**
	(b)	Define hash function. Explain it with suitable example.	**04**
	(c)	Write an algorithm for the following stack operations. 1) PUSH 2) POP 3) DISPLAY	**07**
		OR	
	(c)	Write an algorithm for the following queue operations. 1) INSERT 2) DELETE 3) DISPLAY	**07**
Q.3	**(a)**	Write an algorithm to add a node into a binary search tree.	**03**
	(b)	Explain Dequeue and Priority queue in detail.	**04**
	(c)	Construct the minimum spanning tree using prim's algorithm for the following graph.	**07**



OR

Q.3	**(a)**	Write an algorithm to delete an item from the doubly linked list.	**03**
	(b)	Differentiate: BFS and DFS.	**04**
	(c)	Construct the minimum spanning tree using krushkal's algorithm for the following graph.	**07**



Q.4	**(a)**	Define the terms below: 1) Threaded Tree 2) Acyclic graph 3) Sparse matrix	**03**
	(b)	Explain AVL tree in detail with suitable example.	**04**
	(c)	Construct a binary tree from the traversals given below: Inorder : D, B, E, H, A, I, F, C, G Preorder : A, B, D, E, H, C, F, I, G	**07**

OR

Q.4	**(a)**	Define the terms below: 1) Complete Binary Tree 2) Forest 3) Abstract data type	**03**
	(b)	Explain 2-3 tree in brief.	**04**
	(c)	Write an algorithm to insert an item as below: 1) At the start of the linked list 2) At the end of the linked list	**07**

Q.5	**(a)**	Define file. Explain its types.	**03**
	(b)	Explain bubble sort with suitable example.	**04**
	(c)	Build a chained hash table of 10 memory locations. Insert the keys 121, 3,4, 31, 61, 24, 7, 87, 8, 9 in hash table using chaining. Use h(k) = k mod m (m=10).	**07**

OR

Q.5	**(a)**	Define file organization. Explain different file organizations.	**03**
	(b)	Sort the following array elements using insertion sort algorithm. 8, -2, 5, 3, 9, 4, 6	**04**
	(c)	Explain Dijakstra's shortest path using suitable example.	**07**

****

Q.1 (a) Explain data structure. Enlist and explain the importance of data structure. (3 mark)

Introduction / Definition:

A **data structure** is a systematic way of **organizing, storing, and managing data** in computer memory so that data can be **accessed, processed, and modified efficiently**. It defines the relationship between data elements and the operations that can be performed on them. Proper use of data structures improves **program efficiency, performance, and clarity**.

Need for Data Structure:

- Computers handle **large amounts of data**, and efficient organization is required.
- Different problems require **different ways of storing and accessing data**.
- Data structures help in **optimizing time and memory usage**.

Types of Data Structures (Brief):

- **Primitive Data Structures:** int, float, char, etc.
- **Non-Primitive Data Structures:**
  - Linear: Array, Stack, Queue, Linked List
  - Non-linear: Tree, Graph

Importance of Data Structure:

1. Efficient Data Access:

- Data structures allow **fast access and retrieval** of data.

- Example: Arrays provide direct access using index, BST provides fast searching.

2. Efficient Memory Management:

- Proper data structures reduce **memory wastage**.
- Example: Linked list allocates memory dynamically, unlike arrays.

3. Improves Program Performance:

- Optimized data structures reduce **execution time**.
- Example: Stack and queue operations are efficient for specific applications.

4. Helps in Problem Solving:

- Many real-world problems can be easily solved using suitable data structures.
- Example: Graphs for networks, trees for hierarchical data.

5. Supports Algorithm Design:

- Data structures are the **foundation of algorithms**.
- Efficient algorithms depend on appropriate data structures.

6. Code Reusability and Maintainability:

- Well-structured data makes programs **easy to understand, modify, and maintain**.

Example in Words:

- In a **library management system**, arrays or linked lists store books, queues manage issue requests, and trees organize book categories efficiently.

(b) Define time and space complexity. Derive time complexity of merge sort. (4 mark)

Time and Space Complexity & Time Complexity of Merge Sort

1. Time Complexity

Definition:
Time complexity is a measure of the **amount of time an algorithm takes to execute** as a function of the input size *n*. It helps in analyzing the **efficiency of an algorithm** independent of machine speed.

- Expressed using **Big-O notation** such as O(n), O(log n), O(n log n).
- Indicates how execution time grows with input size.

---

2. Space Complexity

Definition:
Space complexity is the **amount of memory space required by an algorithm** during its execution, including input space and auxiliary (temporary) space.

- It helps in evaluating **memory efficiency** of an algorithm.
- Example: Recursive algorithms usually require extra stack space.

---

3. Time Complexity of Merge Sort

Explanation:
Merge Sort follows the **Divide and Conquer** technique.

- The array is divided into **two equal halves** recursively until single elements remain.
- Each division takes **log n** levels.

- At each level, the **merge operation** takes **O(n)** time to combine elements.

Derivation:

- Dividing the array → $\log_2 n$ levels
- Merging at each level → O(n) time

$$T(n)=n\log n$$

Final Time Complexity of Merge Sort:

- **Best Case:** O(n log n)
- **Average Case:** O(n log n)
- **Worst Case:** O(n log n)

Key Points:

- Merge Sort has **consistent performance** in all cases.
- It is efficient for **large data sets**.
- Requires extra space, so space complexity is **O(n)**.

(c) 1) Evaluate the postfix expression in tabular forms: 2 5 3 - * 8 / 4 + 2) Evaluate the prefix expression in tabular forms: / 7 * 1 + 4 – 6 3(7 mark)

Introduction:

Postfix (Reverse Polish Notation) and **Prefix (Polish Notation)** expressions eliminate the need for parentheses by fixing the position of operators.

- **Postfix:** Operator comes **after operands**
- **Prefix:** Operator comes **before operands**
  Both expressions are efficiently evaluated using a **stack data**

structure, which follows the **LIFO (Last In First Out)** principle.

---

1) Evaluation of Postfix Expression

Given Postfix Expression:

2 5 3 - * 8 / 4 +

Rule for Postfix Evaluation:

- Scan expression **left to right**
- If operand → **push onto stack**
- If operator → **pop two operands**, apply operator
- Push result back to stack

Postfix Evaluation Table:

Step	Symbol	Operation Performed	Stack Content
1	2	Push 2	2
2	5	Push 5	2, 5
3	3	Push 3	2, 5, 3
4	-	5 − 3 = 2	2, 2
5	*	2 × 2 = 4	4
6	8	Push 8	4, 8
7	/	4 ÷ 8 = 0.5	0.5
8	4	Push 4	0.5, 4
9	+	0.5 + 4 = 4.5	4.5

Final Result (Postfix):

$$\boxed{4.5}$$

2) Evaluation of Prefix Expression

Given Prefix Expression:

/ 7 * 1 + 4 - 6 3

Rule for Prefix Evaluation:

- Scan expression **right to left**
- If operand → **push onto stack**
- If operator → **pop two operands**, apply operator
- Push result back to stack

Prefix Evaluation Table:

Step	Symbol	Operation Performed	Stack Content
1	3	Push 3	3
2	6	Push 6	3, 6
3	-	6 − 3 = 3	3
4	4	Push 4	3, 4
5	+	4 + 3 = 7	7
6	1	Push 1	7, 1
7	*	1 × 7 = 7	7
8	7	Push 7	7, 7
9	/	7 ÷ 7 = 1	1

Final Result (Prefix):

1

Key Points:

- Postfix is evaluated **left to right**.
- Prefix is evaluated **right to left**.
- Stack is essential for handling operators and operands.
- These notations are efficient for **expression evaluation in compilers**.

✅ **Final Answers:**

- **Postfix Expression Result: 4.5**
- **Prefix Expression Result: 1**

Q.2 (a) Explain Tower of Hanoi with suitable example. (3 mark)

Tower of Hanoi

Introduction / Definition:
The **Tower of Hanoi** is a classic **recursive problem** used to explain the concept of **recursion** and **divide and conquer technique**. It consists of **three rods** and a number of **disks of different sizes** placed on one rod in increasing order of size, with the smallest disk on top.

Rules of Tower of Hanoi:

1. Only **one disk can be moved at a time**.
2. A disk can be placed **only on top of a larger disk**.
3. All disks must be moved from **source rod** to **destination rod** using an **auxiliary rod**.

Working / Explanation:

- The problem is solved by **recursively moving** $n-1$disks from the source rod to the auxiliary rod.
- Then the largest disk is moved to the destination rod.
- Finally, the $n-1$disks are moved from the auxiliary rod to the destination rod.

Example:
For **3 disks**, the total number of moves required is:

$$2n-1=23-1=7 \text{ moves}$$

This example clearly shows how recursion simplifies the problem.

(b) Define hash function. Explain it with suitable example. (4 mark)

Definition:

A **hash function** is a mathematical function that converts a given **key value** into a **fixed-size integer**, called a **hash value** or **hash address**, which is used as an **index in a hash table**. The main purpose of a hash function is to **store and retrieve data efficiently** in minimum time.

Explanation:

- A hash function takes a **key** as input and computes an **address** where the record will be stored.
- It should distribute keys **uniformly** over the hash table to reduce collisions.
- A good hash function is **easy to compute** and minimizes **hash collisions**.
- Hashing allows **fast search, insertion, and deletion** operations, usually in **O(1)** time.

Common Hash Function Methods:

- **Division Method:** h(k) = k mod m
- **Mid-square Method**
- **Folding Method**

Example:

Consider a hash table of size **10** and the division method hash function:

Insert the following keys:

- Key = 23 → h(23) = 23 mod 10 = 3
- Key = 45 → h(45) = 45 mod 10 = 5
- Key = 36 → h(36) = 36 mod 10 = 6

Each key is stored at the calculated index in the hash table.

Key Points / Advantages:

- Provides **fast access** to data.
- Reduces searching time compared to linear search.
- Widely used in **symbol tables, databases, and password storage**.

(c) Write an algorithm for the following stack operations. 1) PUSH 2) POP 3) DISPLAY (7 mark)

Introduction / Definition:

A **stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle. The element inserted last is the first one to be removed.

Stack operations are performed at one end called the **TOP**. The main stack operations are **PUSH**, **POP**, and **DISPLAY**.

Basic Terms Used:

- **Stack:** Collection of elements
- **TOP:** Pointer/index that indicates the topmost element
- **Overflow:** Stack is full
- **Underflow:** Stack is empty

---

1) PUSH Operation (Insertion into Stack)

Purpose:
To insert an element into the stack at the **top position**.

Algorithm: PUSH(STACK, TOP, MAX, ITEM)

1. If TOP == MAX - 1, then
    - Print **"Stack Overflow"**
    - Exit
2. Else
    - TOP = TOP + 1
    - STACK[TOP] = ITEM
3. End

---

2) POP Operation (Deletion from Stack)

Purpose:
To remove the **topmost element** from the stack.

Algorithm: POP(STACK, TOP)

1. If TOP == -1, then
    - Print **"Stack Underflow"**
    - Exit
2. Else
    - ITEM = STACK[TOP]
    - TOP = TOP - 1
3. End

---

3) DISPLAY Operation (Show Stack Elements)

Purpose:
To display all elements of the stack from **top to bottom**.

Algorithm: DISPLAY(STACK, TOP)

1. If TOP == -1, then
    - Print **"Stack is Empty"**
    - Exit
2. Else
    - For i = TOP down to 0
        - Print STACK[i]
3. End

Example in Words:

- Initially, stack is empty (TOP = -1)
- PUSH 10 → Stack: 10
- PUSH 20 → Stack: 20, 10

- POP → Removes 20 → Stack: 10
- DISPLAY → Output: 10

Key Points / Features:

- Stack follows **LIFO principle**.
- PUSH and POP operations take **O(1) time**.
- Used in **expression evaluation, recursion, undo/redo operations**.

OR

(c) Write an algorithm for the following queue operations. 1) INSERT 2) DELETE 3) DISPLAY (7 mark)

Introduction / Definition:

A **queue** is a linear data structure that follows the **FIFO (First In First Out)** principle. The element that is inserted first is removed first. Queue operations are performed at **two ends**, called **FRONT** (deletion) and **REAR** (insertion). The basic queue operations are **INSERT (Enqueue)**, **DELETE (Dequeue)**, and **DISPLAY**.

Basic Terms Used:

- **Queue:** Collection of elements
- **FRONT:** Points to the first element
- **REAR:** Points to the last element
- **Overflow:** Queue is full
- **Underflow:** Queue is empty

---

1) INSERT Operation (Enqueue)

Purpose:
To insert an element into the queue at the **REAR end**.

Algorithm: INSERT(QUEUE, FRONT, REAR, MAX, ITEM)

1. If REAR == MAX - 1, then
    - Print **"Queue Overflow"**
    - Exit
2. Else if FRONT == -1, then
    - Set FRONT = 0
    - REAR = 0
3. Else
    - REAR = REAR + 1
4. Set QUEUE[REAR] = ITEM
5. End

---

2) DELETE Operation (Dequeue)

Purpose:
To remove an element from the queue at the **FRONT end**.

Algorithm: DELETE(QUEUE, FRONT, REAR)

1. If FRONT == -1 or FRONT > REAR, then
    - Print **"Queue Underflow"**
    - Exit
2. Else

    - ITEM = QUEUE[FRONT]
    - FRONT = FRONT + 1
3. If FRONT > REAR, then
    - Set FRONT = -1
    - Set REAR = -1
4. End

---

3) DISPLAY Operation (Show Queue Elements)

Purpose:
To display all elements of the queue from **FRONT to REAR**.

Algorithm: DISPLAY(QUEUE, FRONT, REAR)

1. If FRONT == -1, then
    - Print **"Queue is Empty"**
    - Exit
2. Else
    - For i = FRONT to REAR
        - Print QUEUE[i]
3. End

Example in Words:

- Initially: Queue empty (FRONT = REAR = -1)
- INSERT 10 → Queue: 10
- INSERT 20 → Queue: 10, 20
- DELETE → Removes 10 → Queue: 20

- DISPLAY → Output: 20

Key Points / Features:

- Queue follows **FIFO principle**.
- INSERT and DELETE operations take **O(1) time**.
- Used in **CPU scheduling, printer queue, and buffering systems**.

OR

(c) Write an algorithm for the following queue operations. 1) INSERT 2) DELETE 3) DISPLAY (7 mark)

Introduction / Definition:

A **queue** is a linear data structure that follows the **FIFO (First In First Out)** principle. The element that is inserted first is removed first. Queue operations are performed at **two ends**, called **FRONT** (deletion) and **REAR** (insertion). The basic queue operations are **INSERT (Enqueue)**, **DELETE (Dequeue)**, and **DISPLAY**.

Basic Terms Used:

- **Queue:** Collection of elements
- **FRONT:** Points to the first element
- **REAR:** Points to the last element
- **Overflow:** Queue is full
- **Underflow:** Queue is empty

---

1) INSERT Operation (Enqueue)

Purpose:
To insert an element into the queue at the **REAR end**.

Algorithm: INSERT(QUEUE, FRONT, REAR, MAX, ITEM)

1. If REAR == MAX - 1, then
    - Print **"Queue Overflow"**
    - Exit
2. Else if FRONT == -1, then
    - Set FRONT = 0
    - REAR = 0
3. Else
    - REAR = REAR + 1
4. Set QUEUE[REAR] = ITEM
5. End

---

2) DELETE Operation (Dequeue)

Purpose:
To remove an element from the queue at the **FRONT end**.

Algorithm: DELETE(QUEUE, FRONT, REAR)

1. If FRONT == -1 or FRONT > REAR, then
    - Print **"Queue Underflow"**
    - Exit
2. Else
    - ITEM = QUEUE[FRONT]
    - FRONT = FRONT + 1

3. If FRONT > REAR, then
    - Set FRONT = -1
    - Set REAR = -1
4. End

---

3) DISPLAY Operation (Show Queue Elements)

Purpose:
To display all elements of the queue from **FRONT to REAR**.

Algorithm: DISPLAY(QUEUE, FRONT, REAR)

1. If FRONT == -1, then
    - Print **"Queue is Empty"**
    - Exit
2. Else
    - For i = FRONT to REAR
        - Print QUEUE[i]
3. End

Example in Words:

- Initially: Queue empty (FRONT = REAR = -1)
- INSERT 10 → Queue: 10
- INSERT 20 → Queue: 10, 20
- DELETE → Removes 10 → Queue: 20
- DISPLAY → Output: 20

Key Points / Features:

- Queue follows **FIFO principle**.
- INSERT and DELETE operations take **O(1) time**.
- Used in **CPU scheduling, printer queue, and buffering systems**.

Q.3 (a) Write an algorithm to add a node into a binary search tree. (3 mark)

Introduction / Definition:
A **Binary Search Tree (BST)** is a binary tree in which, for every node, all values in the **left subtree are smaller** than the node's value and all values in the **right subtree are greater**. Inserting a node in a BST must maintain this property.

Algorithm: INSERT_BST(ROOT, ITEM)

1. If ROOT == NULL, then
    - Create a new node with ITEM
    - Set ROOT = new node
    - Exit
2. If ITEM < ROOT->data, then
    - Insert ITEM into the **left subtree**
3. Else if ITEM > ROOT->data, then
    - Insert ITEM into the **right subtree**
4. End

Key Points:

- Insertion is done by **comparing values**.
- BST property (**Left < Root < Right**) is preserved.
- Time complexity is **O(log n)** for balanced BST and **O(n)** in worst case.

(b) Explain Dequeue and Priority queue in detail. (4 mark)

Dequeue and Priority Queue

1) Dequeue (Double Ended Queue)

Definition:
A **Dequeue (Double Ended Queue)** is a linear data structure in which **insertion and deletion can be performed at both ends**, that is, at the **front** as well as at the **rear**. It is a generalized form of queue.

Types of Dequeue:

- **Input Restricted Dequeue:**
  - Insertion allowed at only one end, deletion at both ends.
- **Output Restricted Dequeue:**
  - Deletion allowed at only one end, insertion at both ends.

Features / Uses:

- Provides **flexibility** in data insertion and deletion.
- Can be implemented using **array or linked list**.
- Used in **job scheduling, palindrome checking, and sliding window problems**.

---

2) Priority Queue

Definition:
A **priority queue** is a special type of queue in which each element is

associated with a **priority value**. Elements are removed based on their **priority**, not strictly by insertion order.

Types of Priority Queue:

- **Max Priority Queue:** Higher priority elements are deleted first.
- **Min Priority Queue:** Lower priority elements are deleted first.

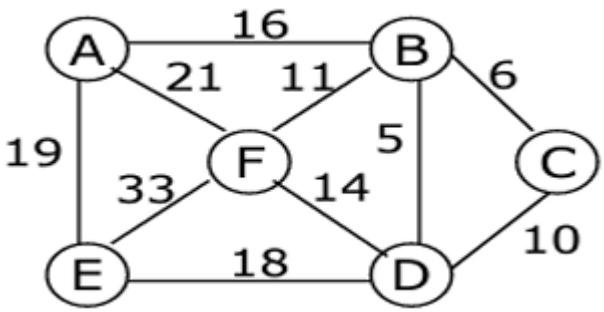
Features / Uses:

- Elements with **same priority** are served in **FIFO order**.
- Implemented using **arrays, linked lists, or heaps**.
- Widely used in **CPU scheduling, Dijkstra's algorithm, and event simulation**.

Key Differences (Brief):

Feature	Dequeue	Priority Queue
Insertion/Deletion	Both ends	Based on priority
Order	FIFO from either end	Priority-based
Application	Scheduling, buffering	OS scheduling, graph algorithms

(c) Construct the minimum spanning tree using prim's algorithm for the following graph. (7 mark)



Introduction / Definition:

A **Minimum Spanning Tree (MST)** of a connected, weighted, undirected graph is a subgraph that:

- Includes **all vertices**,
- Contains **no cycles**, and
- Has the **minimum possible total edge weight**.

Prim's algorithm is a **greedy algorithm** that builds the MST by **starting from any vertex** and repeatedly adding the **minimum weight edge** that connects a vertex in the tree to a vertex outside the tree.

Given Graph (Description):

Vertices: **A, B, C, D, E, F**
Edges with weights (as shown in graph):

- A–B = 16, A–F = 21, A–E = 19
- B–F = 11, B–D = 5, B–C = 6
- F–D = 14, F–E = 33
- D–E = 18, D–C = 10

(The graph shows all vertices connected with given edge weights.)

Prim's Algorithm – Steps:

1. Start with any vertex (assume **A**).
2. Add the **minimum weight edge** connecting the current tree to a new vertex.
3. Repeat until **all vertices are included**.

Step-by-Step Construction (Starting from Vertex A):

Step 1:

- Start with **A**
- Possible edges: A–B (16), A–E (19), A–F (21)
- **Select A–B (16)** → minimum
- MST edges: {A–B}

Step 2:

- Tree vertices: A, B
- Possible edges:
  - B–D (5), B–C (6), B–F (11), A–E (19), A–F (21)
- **Select B–D (5)**
- MST edges: {A–B, B–D}

Step 3:

- Tree vertices: A, B, D
- Possible edges:
  - B–C (6), D–C (10), B–F (11), D–E (18), A–E (19)
- **Select B–C (6)**
- MST edges: {A–B, B–D, B–C}

Step 4:

- Tree vertices: A, B, C, D
- Possible edges:
  - B–F (11), D–F (14), D–E (18), A–E (19)
- **Select B–F (11)**
- MST edges: {A–B, B–D, B–C, B–F}

Step 5:

- Tree vertices: A, B, C, D, F
- Possible edges:
  - D–E (18), A–E (19), F–E (33)
- **Select D–E (18)**
- MST edges: {A–B, B–D, B–C, B–F, D–E}

All vertices are now included.

Final Minimum Spanning Tree Edges:

Edge	Weight
A – B	16
B – D	5
B – C	6
B – F	11
D – E	18

Total Cost of MST:

16+5+6+11+18=56

Key Points / Features of Prim's Algorithm:

- Greedy algorithm
- Always selects **minimum weight edge**
- Time complexity:
  - **$O(V^2)$** using adjacency matrix

    - **O(E log V)** using priority queue
- Suitable for **dense graphs**

Q.3 (a) Write an algorithm to delete an item from the doubly linked list. (3 mark)

introduction / Definition:
A **doubly linked list** is a linear data structure in which each node contains **three parts**: a pointer to the **previous node**, the **data**, and a pointer to the **next node**. Deletion in a doubly linked list is efficient because traversal is possible in both directions.

Algorithm: DELETE_DLL(HEAD, ITEM)

1. If HEAD == NULL, then
    - Print **"List is Empty"**
    - Exit
2. Set TEMP = HEAD.
3. Traverse the list until TEMP->data == ITEM.
4. If TEMP == HEAD, then
    - Set HEAD = TEMP->next
    - If HEAD != NULL, set HEAD->prev = NULL
5. Else if TEMP->next == NULL, then
    - Set TEMP->prev->next = NULL
6. Else
    - Set TEMP->prev->next = TEMP->next
    - Set TEMP->next->prev = TEMP->prev
7. Free memory of TEMP.

8. End

Key Points:

- Deletion handles **three cases**: first node, last node, and middle node.
- Previous and next pointers are updated properly.
- Time complexity is **O(n)**.

(b) Differentiate: BFS and DFS. (4 mark)

Difference between BFS and DFS

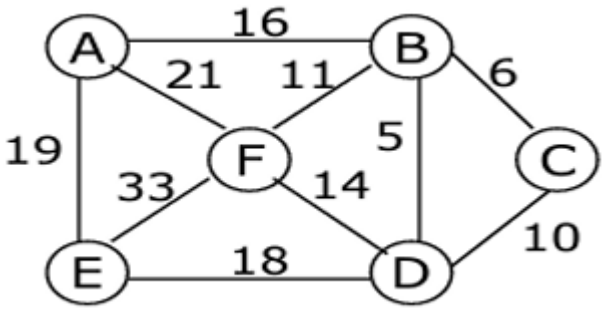
Basis	**Breadth First Search (BFS)**	**Depth First Search (DFS)**
Traversal Method	Traverses graph **level by level**	Traverses graph **depth wise**
Data Structure Used	Uses **Queue**	Uses **Stack** (or recursion)
Path Finding	Finds **shortest path** in unweighted graph	Does **not guarantee shortest path**
Memory Usage	Requires **more memory** to store all neighbors	Requires **less memory**
Completeness	Always complete for finite graphs	May not be complete in infinite graphs
Applications	Shortest path, broadcasting	Topological sort, cycle detection

Key Points:

- BFS explores **all adjacent vertices first**, then moves deeper.

- DFS explores **one path completely before backtracking**.
- BFS is preferred when **minimum distance** is required.
- DFS is preferred for **structure exploration**.

(c) Construct the minimum spanning tree using krushkal's algorithm for the following graph. (7 mark)



Introduction / Definition:

A **Minimum Spanning Tree (MST)** of a connected, weighted, undirected graph is a spanning tree that connects **all vertices without forming cycles** and has the **minimum possible total edge weight**.

Kruskal's algorithm is a **greedy algorithm** that constructs the MST by **sorting all edges in increasing order of weight** and then selecting edges one by one, ensuring that **no cycle is formed**.

Given Graph (Description):

Vertices: **A, B, C, D, E, F**

Edges with weights (as shown in the graph):

- B–D = 5
- B–C = 6
- D–C = 10
- B–F = 11
- F–D = 14

- A–B = 16
- D–E = 18
- A–E = 19
- A–F = 21
- E–F = 33

Steps of Kruskal's Algorithm:

1. List all edges of the graph.
2. Sort edges in **ascending order of weight**.
3. Select the smallest edge that does **not form a cycle**.
4. Repeat step 3 until **(V − 1) edges** are selected, where V is number of vertices.

Sorted Edges in Ascending Order:

Edge	Weight
B – D	5
B – C	6
D – C	10
B – F	11
F – D	14
A – B	16
D – E	18
A – E	19

Edge	Weight
A – F	21
E – F	33

Step-by-Step MST Construction:

1. **Select B–D (5)** → No cycle
   - MST = {B–D}
2. **Select B–C (6)** → No cycle
   - MST = {B–D, B–C}
3. **Consider D–C (10)** → Forms a cycle (B–D–C–B)
   - ❌ Reject
4. **Select B–F (11)** → No cycle
   - MST = {B–D, B–C, B–F}
5. **Consider F–D (14)** → Forms a cycle
   - ❌ Reject
6. **Select A–B (16)** → No cycle
   - MST = {B–D, B–C, B–F, A–B}
7. **Select D–E (18)** → No cycle
   - MST = {B–D, B–C, B–F, A–B, D–E}

Now, **5 edges = V − 1 = 6 − 1**, so MST is complete.

Final Minimum Spanning Tree Edges:

Edge Weight

B – D 5

B – C 6

B – F 11

A – B 16

D – E 18

Total Cost of Minimum Spanning Tree:

5+6+11+16+18=56

Key Features of Kruskal's Algorithm:

- Greedy algorithm
- Uses **edge-based approach**
- Avoids cycles using **disjoint sets / union-find**
- Time complexity:
  - **O(E log E)** due to sorting of edges
- Suitable for **sparse graphs**

Q.4 (a) Define the terms below: 1) Threaded Tree 2) Acyclic graph 3) Sparse matrix (3 mark)

1) Threaded Tree:

A **threaded tree** is a special type of binary tree in which the **NULL pointers** of leaf nodes are replaced by **threads** that point to the **inorder predecessor or inorder successor** of the node. This technique allows **tree traversal without using recursion or a stack**, improving traversal efficiency.

---

2) Acyclic Graph:

An **acyclic graph** is a graph that **does not contain any cycles**.

- In an **undirected graph**, acyclic means no closed path exists.
- In a **directed graph**, it is called a **Directed Acyclic Graph (DAG)**.
  Acyclic graphs are widely used in **scheduling, dependency resolution, and compiler design**.

---

3) Sparse Matrix:

A **sparse matrix** is a matrix in which the **number of zero elements is much greater than the number of non-zero elements**. To save memory, sparse matrices are stored using **special representations** like **3-tuple form** or **linked list representation** instead of a normal 2D array.

Key Point:

These concepts help improve **memory efficiency, traversal performance, and problem modeling** in data structures.

(b) Explain AVL tree in detail with suitable example.(4 mark)

AVL Tree

Definition / Introduction:

An **AVL tree** is a **self-balancing Binary Search Tree (BST)** in which the **height difference between the left and right subtrees** of every node is at most **1**. This height difference is called the **balance factor**.

AVL trees were the first self-balancing BSTs and ensure **efficient searching**.

Balance Factor:

- Valid balance factor values are **−1, 0, +1**.
- If balance factor goes outside this range, the tree becomes **unbalanced**.

Balancing Technique (Rotations):

AVL trees maintain balance using **tree rotations**:

- **LL Rotation (Right Rotation)**
- **RR Rotation (Left Rotation)**
- **LR Rotation (Left-Right Rotation)**
- **RL Rotation (Right-Left Rotation)**

These rotations rearrange nodes to restore balance.

Example (Insertion):

Insert the elements **10, 20, 30** into an AVL tree:

- Insert 10 → root
- Insert 20 → right child of 10
- Insert 30 → right child of 20 → imbalance occurs
- Perform **RR rotation**, resulting tree has **20 as root**, 10 as left child, and 30 as right child

This restores the balance factor.

Advantages / Key Points:

- Guaranteed **O(log n)** time for search, insertion, and deletion
- Prevents tree from becoming skewed
- In-order traversal gives **sorted order**

(c) Construct a binary tree from the traversals given below: Inorder : D, B, E, H, A, I, F, C, G Preorder : A, B, D, E, H, C, F, I, G (7 mark)

We can construct the binary tree **step by step** using the given **inorder** and **preorder** traversals. Let's solve it carefully.

Given:

- Inorder: D, B, E, H, A, I, F, C, G
- Preorder: A, B, D, E, H, C, F, I, G

Step 1: Identify the root

- **Preorder** starts with the root.
- Root = A

Step 2: Split inorder into left and right subtrees

- In **inorder**, elements to the left of A belong to the **left subtree**, elements to the right belong to the **right subtree**.

Inorder left of A: D, B, E, H
Inorder right of A: I, F, C, G

Step 3: Preorder for left and right subtrees

- **Preorder** sequence (after root A) is divided according to the inorder split.

Left subtree preorder = elements in preorder that appear in left inorder: B, D, E, H
Right subtree preorder = elements in preorder that appear in right inorder: C, F, I, G

Step 4: Construct left subtree

Inorder left: D, B, E, H
Preorder left: B, D, E, H

1. Root of left subtree = first in preorder left = B
2. In inorder left, B splits:
    - Left: D
    - Right: E, H

Left child of B: D
Right child of B: subtree with

- Inorder: E, H
- Preorder: E, H

3. Root of this subtree = E

- Left: none
- Right: H

So left subtree becomes:

Step 5: Construct right subtree

Inorder right: I, F, C, G
Preorder right: C, F, I, G

1. Root of right subtree = first in preorder right = C
2. In inorder right, C splits:
    - Left: I, F
    - Right: G

Right child of A = C

- Left subtree: Inorder: I, F, Preorder: F, I
- Right subtree: Inorder: G, Preorder: G

Left child of C:

- Root = F
- Inorder split: Left = I, Right = none
- So F has left child = I, right = null

Right child of C: G

Right subtree:

```
    C
   / \
  F   G
 /
I
```

Step 6: Combine with root A

Wait, let's check carefully:

- Left subtree B is D left, E right, and E has H right → ✅
- Right subtree C is F left with I left → Actually I comes before F?

Let's double-check **right subtree**:

- Inorder right: I, F, C, G
- Preorder right: C, F, I, G
- Root = C

Left of C in inorder = I, F

- Preorder left of C = F, I

Now construct subtree:

- Preorder left of C = F, I
- Root = F
- In inorder of left (I, F) → F splits: Left = I, Right = none

So left child of C = F

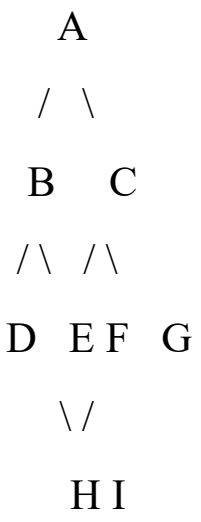
- F's left = I

- F's right = null

Right of C in inorder = G → right child = G ✅

Perfect.

✅ **Final Binary Tree:**



To be exact, structure:

- **A**: left = B, right = C
- **B**: left = D, right = E
- **E**: right = H
- **C**: left = F, right = G
- **F**: left = I

OR

Q.4 (a) Define the terms below: 1) Complete Binary Tree 2) Forest 3) Abstract data type (3 mark)

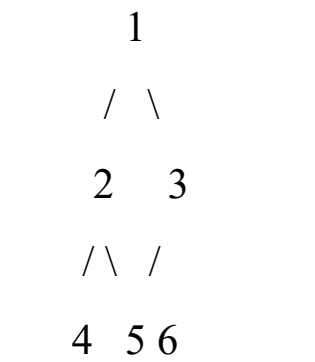
1) Complete Binary Tree

A **complete binary tree** is a binary tree in which:

- All levels **except possibly the last** are completely filled, and

- All nodes in the last level are filled from **left to right** without any gaps.

Example:



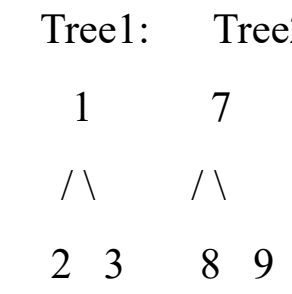
---

2) Forest

A **forest** is a collection of **one or more disjoint trees**.

- If you have multiple trees, together they are called a forest.
- A single tree is also considered a forest with just one tree.

Example:



Together, Tree1 and Tree2 form a **forest**.

---

3) Abstract Data Type (ADT)

An **Abstract Data Type (ADT)** is a **logical description** of a data structure, specifying:

- **What operations** can be performed on the data (like insert, delete, traverse), and

- **What behavior** these operations exhibit, without specifying **how these operations are implemented**.

Example:

- Stack ADT: supports push(), pop(), peek().
- Queue ADT: supports enqueue(), dequeue().

(b) Explain 2-3 tree in brief. (4 mark)

2-3 Tree

A **2-3 tree** is a **self-balancing search tree** in which every **internal node** can have either **2 or 3 children**, and all **leaves are at the same level**.

Properties:

1. **2-node:** Contains **1 key** and has **2 children**.
2. **3-node:** Contains **2 keys** and has **3 children**.
3. The tree **remains balanced** after insertions and deletions.
4. Keys in nodes are arranged **in sorted order**, ensuring **efficient search, insertion, and deletion**.

Example:

```
      [10 | 20]
     /    |    \
  [5]   [15]   [25]
```

- Root [10 | 20] is a 3-node (2 keys, 3 children).
- Leaf nodes [5], [15], [25] are 2-nodes (1 key each).

Advantages:

- Guarantees **O(log n)** time for search, insert, and delete.
- Automatically **keeps the tree balanced**.

(c) Write an algorithm to insert an item as below: 1) At the start of the linked list 2) At the end of the linked list (7 mark)

Algorithm 1: Insert at the Start of a Linked List

Input: head pointer of the linked list, item to insert
Output: Updated linked list with item at the start

Steps:

1. **Create a new node** newNode
2. Set newNode.data = item
3. Set newNode.next = head
4. Set head = newNode
5. **End**

Explanation:

- The new node points to the current head.
- Head is updated to the new node.

Flow:

head -> 10 -> 20 -> 30

Insert 5 at start

head -> 5 -> 10 -> 20 -> 30

Algorithm 2: Insert at the End of a Linked List

Input: head pointer of the linked list, item to insert
Output: Updated linked list with item at the end

Steps:

1. **Create a new node** newNode
2. Set newNode.data = item

3. Set newNode.next = NULL
4. If head == NULL
    - Set head = newNode
    - **Exit**
5. Else
    - Set temp = head
    - Traverse the list until temp.next == NULL
    - Set temp.next = newNode
6. **End**

Explanation:

- Traverse to the last node.
- Append the new node at the end.

Flow:

head -> 10 -> 20 -> 30

Insert 40 at end

head -> 10 -> 20 -> 30 -> 40

Q.5 (a) Define file. Explain its types. (3 mark)

Definition of File:

A **file** is a **collection of related data** stored permanently on secondary storage (like a hard disk or SSD).

- Files provide a way to **store and retrieve data** even after the program ends.
- Each file has a **name** and **type/extension** that identifies its contents.

Types of Files:

1. **Text File:**
    - Stores data in **human-readable form** (characters).
    - Each line is a sequence of characters ending with a newline.
    - Example: .txt, .csv, .html
2. **Binary File:**
    - Stores data in **machine-readable form** (binary codes).
    - Cannot be directly read by humans.
    - Example: .exe, .jpg, .mp3
3. **Sequential File:**
    - Data is stored and accessed **in order**, one record after another.
    - Suitable for batch processing.
4. **Random/Direct Access File:**
    - Data can be accessed **directly at any position** without reading the previous data.
    - Suitable for applications like databases.

(b) Explain bubble sort with suitable example. (4 mark)

Definition:
Bubble sort is a **simple comparison-based sorting algorithm** in which **adjacent elements are repeatedly compared and swapped** if they are in the wrong order.

- This process continues until the list is completely sorted.
- It is called "bubble sort" because **larger elements "bubble" to the end** with each pass.

Algorithm (Brief):

1. Start from the first element of the array.
2. Compare the current element with the next element.
3. If the current element is greater than the next, **swap** them.
4. Move to the next element and repeat step 2 until the end of the array.
5. Repeat the above steps for all elements until no swaps are needed.

Example:

Array: [5, 3, 8, 4]

Pass 1:

- Compare 5 & 3 → swap → [3, 5, 8, 4]
- Compare 5 & 8 → no swap → [3, 5, 8, 4]
- Compare 8 & 4 → swap → [3, 5, 4, 8]

Pass 2:

- Compare 3 & 5 → no swap → [3, 5, 4, 8]
- Compare 5 & 4 → swap → [3, 4, 5, 8]
- Compare 5 & 8 → no swap → [3, 4, 5, 8]

Pass 3:

- Compare 3 & 4 → no swap
- Compare 4 & 5 → no swap
- Array is sorted: [3, 4, 5, 8]

Time Complexity:

- Best case: **O(n)** (already sorted)
- Worst case: **O(n²)**

(c) Build a chained hash table of 10 memory locations. Insert the keys 121, 3,4, 31, 61, 24, 7, 87, 8, 9 in hash table using chaining. Use h(k) = k mod m (m=10). (7 mark)

Given:

- **Keys:** 121, 3, 4, 31, 61, 24, 7, 87, 8, 9
- **Hash function:** $h(k) = k \bmod 10$
- **Memory locations:** 0–9
- **Collision handling:** Chaining (linked list at each slot)

Step 1: Compute hash values

Key	h(k) = k mod 10	Slot
121	121 % 10 = 1	1
3	3 % 10 = 3	3
4	4 % 10 = 4	4
31	31 % 10 = 1	1
61	61 % 10 = 1	1
24	24 % 10 = 4	4
7	7 % 10 = 7	7
87	87 % 10 = 7	7
8	8 % 10 = 8	8

Key	h(k) = k mod 10	Slot
9	9 % 10 = 9	9

Step 2: Insert keys using chaining

- **Slot 0:** empty → -
- **Slot 1:** 121 → 31 → 61 → 121 → 31 → 61
- **Slot 2:** empty → -
- **Slot 3:** 3 → 3
- **Slot 4:** 4 → 24 → 4 → 24
- **Slot 5:** empty → -
- **Slot 6:** empty → -
- **Slot 7:** 7 → 87 → 7 → 87
- **Slot 8:** 8 → 8
- **Slot 9:** 9 → 9

Step 3: Final Chained Hash Table

Slot	Keys (Chained)
0	-
1	121 → 31 → 61
2	-
3	3
4	4 → 24

Slot	Keys (Chained)
5	-
6	-
7	7 → 87
8	8
9	9

OR

Q.5 (a) Define file organization. Explain different file organizations. (3 mark)

Definition of File Organization:

File Organization is the **way records are stored in a file** on secondary storage so that they can be efficiently accessed, inserted, deleted, or updated.

- It defines the **physical arrangement** of data and the **method of accessing** it.

Types of File Organization:

1. **Sequential File Organization:**
    - Records are stored **one after another in order**, usually sorted by a key field.
    - Access is **sequential**; to find a record, you may need to traverse from the beginning.
    - Suitable for **batch processing**.
    - Example: Payroll or bank statements.
2. **Random (Direct) File Organization:**

- Records can be accessed **directly using a key** without reading all previous records.
- Uses a **hash function** or address calculation.
- Suitable for applications requiring **fast access**, like databases.

3. **Indexed File Organization:**
   - Uses an **index table** to keep track of record locations.
   - Allows **faster search** without scanning the entire file.
   - Combines benefits of sequential and direct access.

(b) Sort the following array elements using insertion sort algorithm. 8, -2, 5, 3, 9, 4, 6 (4 mark)

Given Array:

8, -2, 5, 3, 9, 4, 6

Insertion Sort:

- Pick elements one by one from the unsorted part and **insert into the correct position** in the sorted part.

Step 1: Initial array

[8 | -2, 5, 3, 9, 4, 6]

- Sorted part: [8]
- Next element: -2 → insert before 8

Array after Step 1:
[-2, 8, 5, 3, 9, 4, 6]

Step 2: Next element = 5

- Compare with 8 → 5 < 8 → shift 8 to right
- Compare with -2 → 5 > -2 → insert after -2

Array after Step 2:
[-2, 5, 8, 3, 9, 4, 6]

Step 3: Next element = 3

- Compare with 8 → 3 < 8 → shift 8
- Compare with 5 → 3 < 5 → shift 5
- Compare with -2 → 3 > -2 → insert after -2

Array after Step 3:
[-2, 3, 5, 8, 9, 4, 6]

Step 4: Next element = 9

- Compare with 8 → 9 > 8 → insert after 8

Array after Step 4:
[-2, 3, 5, 8, 9, 4, 6]

Step 5: Next element = 4

- Compare with 9 → shift 9
- Compare with 8 → shift 8
- Compare with 5 → shift 5
- Compare with 3 → 4 > 3 → insert after 3

Array after Step 5:
[-2, 3, 4, 5, 8, 9, 6]

Step 6: Next element = 6

- Compare with 9 → shift 9
- Compare with 8 → shift 8
- Compare with 5 → 6 > 5 → insert after 5

Array after Step 6:
[-2, 3, 4, 5, 6, 8, 9]

✅ **Sorted Array:**

[-2, 3, 4, 5, 6, 8, 9]

(c) Explain Dijakstra's shortest path using suitable example. (7 mark)

Dijkstra's Algorithm

Definition:
Dijkstra's algorithm finds the **shortest path from a single source vertex** to all other vertices in a **weighted graph** (with **non-negative edge weights**).

Algorithm Steps:

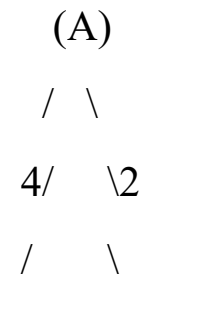
1. **Initialize:**
    - Set the distance to the source vertex S = 0.
    - Set the distance to all other vertices = ∞ (infinity).
    - Mark all vertices as **unvisited**.
2. **Select Vertex:**
    - Pick the unvisited vertex with the **smallest distance** (let's call it U).
3. **Update Distances:**

- For each neighbor V of U, check if the distance from the source via U is **smaller** than the current distance of V.
- If yes, update V's distance.

4. **Mark Visited:**
   - Mark U as visited. A visited vertex will **not be checked again**.
5. **Repeat:**
   - Repeat steps 2–4 until **all vertices are visited**.

Example:

Graph:



- **Vertices:** A, B, C
- **Edges:** A-B=4, A-C=2, B-C=5
- **Source:** A

Step 1: Initialize distances:

A = 0, B = ∞, C = ∞

Step 2: Select A (distance 0):

- Update neighbors:

- B: min(∞, 0+4) = 4
- C: min(∞, 0+2) = 2

Distances: A=0, B=4, C=2

Visited: A

Step 3: Select C (distance 2):

- Update neighbors:
  - B: min(4, 2+5) = 4 → no change

Distances: A=0, B=4, C=2

Visited: A, C

Step 4: Select B (distance 4):

- No neighbors to update

Distances: A=0, B=4, C=2

Visited: A, B, C

Result:

- Shortest distances from **A**:

A → A: 0

A → B: 4

A → C: 2

Key Points:

- Works only with **non-negative weights**.
- Time complexity: **O(V²)** (can be improved using priority queue).
- Produces **shortest path tree** from the source.